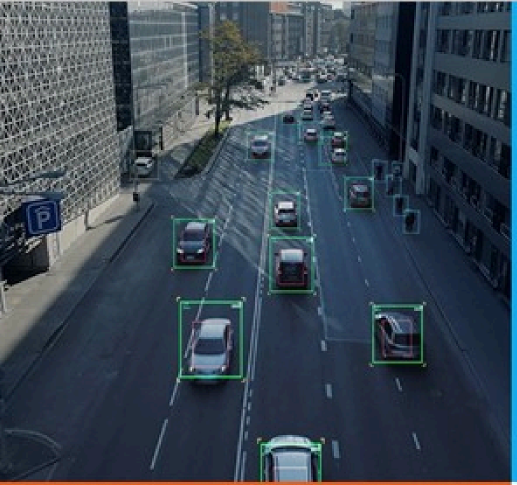


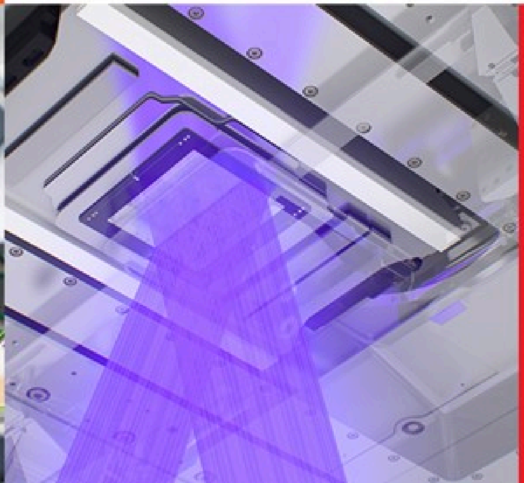
MECHATRONICS

MATHWARE

ASSEMBLY

SIOUX TECHNOLOGIES

ELECTRONICS



SOFTWARE

C++ Training

Strong types

Strong types

```
void create(double mass, double length);
double mass;
double length;
create(mass, length);
create(length, mass);
```

- Typically a developer is not interested if an object stores its mass as double, float or int.
- We assume the library works and just use the API.

Type aliases are not enforced by compiler, only syntactic. But an improvement nonetheless.

```cpp
using Mass=double;
using Length =double;
void create(Mass mass, Length length);
Mass mass;
Length length;
create(mass, length);
create(length, length);
```

- A STRONG type is a type used in place of another type to carry specific meaning through its name.
- Not an alias or typedef

```
struct Mass{};
struct Length{};
void create(Mass mass, Length length);
```

```cpp
template <typename T, typename Tag>
class StrongType
{
public:
    explicit StrongType(T v):_v(std::move(v)){}
private:
    T _v;
};
using Mass = StrongType<double, struct MassTag>;
using Length = StrongType<double, struct LengthTag>;
```

- Conversion between units of the same tag can be implemented using std::ratio.
- std::chrono library to support custom ratios as a template parameter.
- There's a compiler in C++, use it as much as you can.
- You can create custom ratios but the SI ratios are provided by the standard library, for example std::kilo

```cpp
template <typename T, typename Tag, typename Ratio>
class StrongType
{
public:
    explicit StrongType(T v):_v(std::move(v)){}

    template <typename OtherRatio>
    operator StrongType<T, Tag, OtherRatio>() const
    {
        return StrongType<T, Tag, OtherRatio>(_v * Ratio::num / Ratio::den
            * OtherRatio::den / OtherRatio::num);
    }
    friend std::ostream& operator<<(std::ostream& os, StrongType const& x)
    {
        os << x._v;
        return os;
    }
private:
    T _v;
};
```

Curiously recurring template pattern (CRTP)

- For some types it could make sense to add/substract/multiply them, while for others it may not.
- Some applications may have no use for multiplied masses -kg^2 - in which case, multiplication should be blocked.

Slides simplified by removing ratio

```cpp
template <typename StrongType>
struct Crtp
{
    StrongType const& Underlying() const { return static_cast<StrongType const&>(*this);}
};
template <typename StrongType>
struct Addable: Crtp<StrongType>
{
    StrongType operator+(StrongType const& other) const
    {
        return StrongType(this->Underlying().get() + other.get());
    }
};
```

```
template <typename T, typename Tag, template<typename> class... Skills>
class StrongType: public Skills<StrongType<T, Tag, Skills...>>...
{
public:
    explicit StrongType(T v):_v(std::move(v)){}

    T get() const{return _v;}
    friend std::ostream& operator<<(std::ostream& os, StrongType x)
    {
        os << x._v;
        return os;
    }
private:
    T _v;
};
using Mass= StrongType<double, struct MassTag, Addable>;
```

CRTP: Pass derived class as template parameter to base class

```cpp
#include "strongType.h"

int main()
{
    Mass x(10);
    Mass y(4);

    std::cout << x+y<<std::endl;
}
```

Do not reinvent the wheel.

boost::units is BIS when working with units.

```
using namespace boost::units;
using namespace boost::units::si;
quantity<energy> work(const quantity<force>& F, const quantity<length>& dx)
{
    return F*dx; // Defines the relation: work = force * distance.
}
int main()
{
    quantity<force> F(2.0 * newton);
    quantity<length> dx(2.0 *meter);
    quantity<energy> E(work(F,dx));
    std::cout<<"F  = " << F << std::endl <<"dx = " << dx << std::endl <<"E  = " << E << std::endl <<std::endl;
    return 0;
}
---
F  = 2 N
dx = 2 m
E  = 4 J
```

```
using namespace boost::units;
using namespace boost::units::si;
quantity<length> work(const quantity<force>& F, const quantity<length>& dx)
{
    return F*dx;
}
---
<source>:15:13: error: could not convert 'boost::units::operator*(const quantity<Unit1, X>&, const
quantity<Unit2, Y>&) [with Unit1 = unit<list<dim<length_base_dimension, static_rational<1> >,
list<dim<mass_base_dimension, static_rational<1> >, list<dim<time_base_dimension, static_rational<-2> >,
dimensionless_type> > >, homogeneous_system<list<si::meter_base_unit,
list<scaled_base_unit<cgs::gram_base_unit, scale<10, static_rational<3> > >, list<si::second_base_unit,
list<si::ampere_base_unit, list<si::kelvin_base_unit, list<si::mole_base_unit,
list<si::candela_base_unit, list<angle::radian_base_unit, list<angle::steradian_base_unit,
dimensionless_type> > > > > > > > > >; Unit2 = unit<list<dim<length_base_dimension, static_rational<1>
>, dimensionless_type>, homogeneous_system<list<si::meter_base_unit,
list<scaled_base_unit<cgs::gram_base_unit, scale<10, static_rational<3> >
```

exercises

strong_types/ex1.cpp

strong_types/ex2.cpp

strong_types/ex3.cpp

We bring **high-tech** to life

www.sioux.eu